Month 200X, Vol.21, No.X, pp.XX–XX *J. Comput. Sci. & Technol.*

NuMDG: A New Tool for Multiway Decision Graphs Construction

Sa'ed Abed[1], Yassine Mokhtari[2], Otmane Ait Mohamed[2] and Sofine Tahar[2]

[1] *Computer Engineering Department, Hashemite University, Jordan*

E-mail: sabed@hu.edu.jo

[2] *Electrical and Computer Engineering Department, Concordia University, Canada*

E-mail: {mokhtari,ait,tahar}@ece.concordia.ca

Received MONTH DATE, YEAR.

Abstract Multiway Decision Graphs (MDGs) are a canonical representation of a subset of many-sorted first-order logic. This subset generalizes the logic of equality with abstract types and uninterpreted function symbols. The distinction between abstract and concrete sorts mirrors the hardware distinction between data path and control. Here we consider ways to improve MDGs construction. Efficiency is achieved through the use of the Generalized-If-Then-Else (GITE) commonly operator in Binary Decision Diagram packages. Consequently, we review the main algorithms used for MDGs verification techniques. In particular, Relational Product and Pruning by Subsumption. Theses algorithms are defined uniformly through this single GITE operator which will lead to a more efficient implementation. Moreover, we provide their correctness proof. This work can be viewed as a way to accommodate the ROBBD algorithms to the realm of abstract sorts and uninterpreted functions. The new tool, called NuMDG, accepts an extended SMV language, supporting abstract data sorts. Finally, we present experimental results demonstrating the efficiency of the NuMDG tool and evaluating its performance using a set of benchmarks from the SMV package.

Keywords Infinite state model checking, Multiway Decision Graphs, Uninterpreted Functions

1 Introduction

The recent complexity of semiconductor circuits has severely increased the cost for design verification. In addition to the conventional simulation technology, formal verification has

become applicable to real-size designs. Formal verification technology enables us to check the behaviors of designs against given specifications exhaustively. However, formal verification still suffers from intrinsic high computational costs for accomplishing its task. In order to circumvent this difficulty, a method based on datapath abstraction has been proposed.

Binary Decision Diagrams (BDDs) [6] are one of the biggest breakthroughs in computer-aided design in the last decade. BDDs are a canonical and efficient way to represent and manipulate Boolean functions and have been successfully used in numerous applications and improve the capacity of the model checker. BDDs have several useful properties. The representation of many common functions using BDDs is small. The algorithms to handle BDDs are simple. Also a function can be evaluated in linear time in the number of variables and also can be existentially or universally quantified (Boolean) variables in time quadratic in the size of the BDD. Moreover, the order in which the variables appear can be fixed and hence the BDD is a canonical representation for Boolean function. Most BDD packages provide an efficient implementation based on recursive operations using a three operand function commonly known as If-Then-Else (ITE) formulae. Also, they provide many operations that are extensively used in automated verification methods. Unfortunately, their power is mostly restricted to propositional logic, which is often not sufficiently expressive. Moreover, these methods suffer from the drawback that they require a binary representation of the circuit. Every individual bit of every data signal must be encoded by a separate Boolean variable, while the size of ROBDD grows, sometimes exponentially, with the number of variables. This leads to a state explosion problem when ROBDD-based methods are applied to circuits with complex datapath.

To deal with the state explosion problem of traditional Binary Decision Diagram (BDD) based model checking methods, a Multiway Decision Graph (MDG) based model checking approach was proposed in 1997 [11]. MDG is an extended BDD-like data structure with arbitrary number of children for each node and with much more powerful labeling capability for both the nodes and the edges. BDDs can be viewed as a special case of MDGs. In the MDG-based approach, data signals are denoted by abstract variables instead of Boolean variables, and data operators are represented by uninterpreted or partially interpreted function symbols instead of Boolean functions. Thus, the verification can be carried out independently of data path width, which therefore can effectively alleviate the state explosion problem [24]. In MDG-based verification, abstract description of states machines (ASM) are used for modeling systems. In contrast to ordinary Finite State Machines (FSM), the ASM supports non-finite state machines as models in addition to their intended interpretations. The intent is to rise

the abstraction level of automated verification methods to approach those of interactive theorem proving methods without sacrificing automation. MDGs have been investigated from different angles and it culminated in a MDG tool providing Prolog-style MDG-HDL for modeling and different verification techniques including sequential and combinational equivalence checking, invariant checking and a subset of first-order LTL model checking [28, 29]. This work can be viewed as a way to accommodate the ROBBD algorithms to the realm of abstract sorts and uninterpreted functions.

The work presented here mainly improves upon the previous work [11] in one respect. The set of basic operations on MDGs was implemented separately, while ROBDD operations are implemented using a single generic algorithm ITE. This is because the two edges that issue from an ROBDD node labeled x span the ranges of values $\{\mathsf{F}, \mathsf{T}\}$ that x can take, and this makes it possible to reason by case analysis. Consequently, MDGs do not enjoy this property due to abstract variables. The GITE operation can be considered to be a functionally complete three-input logic gate that implements the expression $GITE = (P \wedge Q) \vee (\neg P \wedge H)$. If P is an abstract variable, then there is no MDG representing the formula $\neg P$. In this paper, we claim that it is possible to use the GITE operation to produce an MDG R that is logically equivalent to $(P \wedge Q) \vee (\neg P \wedge H)$ except for some cases that will be discussed later. This leads to improve the efficiency of the existing basic MDG algorithms.

Finally, the work here is an extension to the work presented in [19] in that we provide the correctness proof of all our frame algorithms and implement the tool. We also support our new tool by experimental results executed on different benchmarks from the SMV package. The goal here is to build a robust model checking tool that accepts an extended SMV input language and supports an abstraction mechanism through abstract sorts and uninterpreted functions. Indeed, the results of our prototype shows that such an implementation offers a considerable gain compared to the SMV model checking tool in terms of the size of the MDG transition relation. However, more work should be spent in developing the tool in order to enhance the performance.

The structure of this paper is as follows: Section 2 reviews the closest related work. Section 3 introduces a subset of many-sorted first-order logic that gives MDGs their meaning. Section 4 describes basic MDG algorithms, their optimization and their correctness proof. Section 5 introduces the architecture of NuMDG tool and describes some experimental results. Finally, Section 6 concludes the paper and gives some future research directions.

2 Related Work

In using the logic of equality with uninterpreted functions to verify hardware systems, specific characteristics of the formula describing the correctness condition can be exploited when deciding its validity. Approaches that capture non-finite aspects of the system, by using uninterpreted functions or similar notion like first-order formulae with quantification, are more closely related work.

In Fontaine and Gribomont [13], a BDD-based approach for the combination of theories is presented. It is noted that BDDs, when they are used for first order logic, are not canonical representations any more. For example, BDDs representing $(x \approx y) \wedge p(x)$ and $(x \approx y) \wedge p(y)$ are different although they are logically equivalent. Special constraints have to be added to remove unsatisfiable paths. Then, Goel et al. [14] proposed to decide equality logic formulae by replacing all equalities with new propositional variables, i.e. to replace an equality $v_i \approx v_j$ with a new variable e_{ij}. In this approach the BDD for the resulting formula is calculated without taking into account the transitivity of equalities, and for assignments satisfying the BDD, it is inspected on whether they also satisfy the original equality logic formula.

Burch and Dill [16] have proposed a verification method that uses propositional logic, extended with uninterpreted functions, uninterpreted predicates, and the testing of equality to denote data operations and a decision procedure as a theorem-proving search method. Compared to MDG, their approach does not support representation of a set of states, fixpoint calculation and the transition relation can be applied only a given number of times. Burch and Dills work has generated considerable interest in the use of uninterpreted functions to abstract data operations in processor verification. A common theme has been to adopt Boolean methods in two respects: integration of uninterpreted functions into a symbolic model checkers [12, 4] or developing BDD-based decision procedures [15, 14].

More recently, Bryant *et al.* [5] translate a formula with uninterpreted functions to propositional formula within the theory of equality while preserving validity. Therefore, the resulting formula can be checked efficiently either by a BDD or SAT solver. Later, as found in [27], the new efficient SAT solvers would not have scaled for solving the Boolean formulae if not for the property of Positive Equality that results in at least five orders of magnitude speedup when formally verifying dual-issue superscalar processors with realistic features. Efficient translations from propositional logic to CNF [26], exploiting the special structure of logic of Equality with Uninterpreted Functions and Memories (EUFM), formulae produced with the modeling restrictions, resulted in additional speedup of two orders of magnitude. This reduction is based on Ackermann's approach [1] that consists of replacing each occurrence of a function with a new (domain) variable and adding func-

tional consistency constraints in the formula. The technique also exploits the polarity of equations in the formula to restrict the range allocation. A similar approach is also proposed by Pnueli *et al.* [21] where the key differences are emphasized in [5]. Rodeh *et al.* [23] have used the function elimination method of Bryant *et al.* [7] to further restrict the domain size of the variables using the algorithm in [21]. Shuvendu *et al.* [18] present a generalization of positive equality analysis of Bryant [7], which allows the decision procedure to exploit positive equality in situations where previous approach fails. The new version called robust positive equality, restricts the interpretations to consider in deciding forumals in Equality with Uninterpreted Functions (EUF) to a subset of interpretations considered by the previous approach.

Partial order reduction takes advantages of the fact that, in many cases, when components of a system are not tightly coupled, different execution orders of actions or transitions of different components may result in the same global state. Then, under some conditions [20, 17], in particular, when the interim global states are not relevant to the query being checked, model checkers only need to explore one of the possible execution orders. This may radically reduce model checking complexity.

These approaches are applicable when data operations can be viewed as black-boxes, i.e., the correctness of the system being modeled does not depend on the meaning of these operations. This is usually the form of RTL designs generated by high-level synthesis algorithms that schedule and allocate data operations without being concerned with the specific nature of the operations. However, ignoring properties of data operations leads sometimes to false negatives. For example, a multiplier can be abstracted away when one of its inputs is 0 or 1. In MDG, a simple rewriting system is used to deal with such cases. In [25], Velev combines rewriting rules and Burch and Dill's method [16] to verify out-of-order processors that have a Reorder Buffer.

3 Multiway Decision Graphs Overview

3.1 Sorted Signature

A sorted signature $\Sigma(\mathcal{V}, \mathcal{L}, \mathcal{S})$ consists of an infinite set of variables $\mathcal{V}$, partitioned into a set $\mathcal{V}_{abs}$ of abstract variables and a set $\mathcal{V}_{con}$ of concrete variables, a set of symbols $\mathcal{L}$, partitioned into a set $\mathcal{L}_{CO}$ of cross-operators and a set $\mathcal{L}_F$ of function symbols and a set of sort symbols $\mathcal{S}$, partitioned into a set $\mathcal{S}_{con}$ of concrete sorts and a set $\mathcal{S}_{abs}$ of abstract sorts. All these sets are disjoint. Furthermore there is:

- An arity function that associates to each symbol in $\mathcal{L}$ a natural number. Constant symbols are 0-ary function symbol.

- A function $\eta : \mathcal{V} \to \mathcal{S}$ which gives a sort to each variable symbol.
- A set of sort declarations for terms. A sort declaration for a term is a tuple $t : S$, where t is a non-variable term and $S \in \mathcal{S}_{abs}$ is a sort symbol. We sometimes abbreviate sort declaration $f(x_1, \ldots, x_n) : S$ as $f : S_1 \times \ldots \times S_n \to S$ where S_i is the sort of the variable x_i.
- A set of sort declaration for cross-operators. A sort declaration for a cross-operator is of the form $p : S_1 \times \ldots \times S_n \to S$ where the S_i are sorts and $S \in \mathcal{S}_{con}$

3.2 Well Sorted Terms

The set of well sorted terms $\mathcal{T}(\Sigma, S)$ of sort S in signature Σ is the smallest set such that:

- $x \in \mathcal{T}(\Sigma, S)$ if $x \in \mathcal{V}$ and $\eta(x) \in S$
- $f(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma, S)$ if $t_i \in \mathcal{T}(\Sigma, S_i)$ for $i = 1, \ldots, n$ and $f : S_1 \times \ldots \times S_n \to S$ is a term sort declaration in Σ

The set $\mathcal{T}(\Sigma)$ of all well sorted terms is defined as the union $\bigcup\{\mathcal{T}(\Sigma, S) : S \in \mathcal{S}\}$. If $\mathcal{V} = \emptyset$, then $\mathcal{T}_G(\Sigma, S)$ denotes a set of ground terms i.e. terms that are not containing variables. A substitution σ is represented as a set $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ where $\mathsf{Dom}(\sigma) = \{x_1, \ldots, x_n\}$ and is defined on terms as usual. Its extension by another substitution σ', written $\sigma \oplus \sigma'$, is another substitution such that:

- $\mathsf{Dom}(\sigma) \cap \mathsf{Dom}(\sigma') = \emptyset$ and
- for every variable $x \in \mathsf{Dom}(\sigma \oplus \sigma')$:

$$(\sigma \oplus \sigma')(x) = \begin{cases} \sigma(x) & \text{if } x \in \mathsf{Dom}(\sigma) \\ \sigma'(x) & \text{if } x \in \mathsf{Dom}(\sigma') \end{cases}$$

3.3 Well Formed Directed Formulae (DFs)

The set of well formed formulae $\mathcal{F}(\Sigma, S)$ of sort S in signature Σ is the smallest set such that:

- $x = t$ if $x \in \mathcal{T}(\Sigma, S)$, $t \in \mathcal{T}_G(\Sigma, S)$ and $S \in \mathcal{S}_{con}$.
- $x = t$ if $x, t \in \mathcal{T}(\Sigma, S)$ and $S \in \mathcal{S}_{abs}$.
- $p(t_1, \ldots, t_n) = t$ if $p : S_1 \times \ldots \times S_n \to S$ is a cross-operator declaration in Σ, either $t_i \in \mathcal{T}(\Sigma, S_i)$ and $S_i \in \mathcal{S}_{abs}$ or $t_i \in \mathcal{T}_G(\Sigma, S_i)$ and $S_i \in \mathcal{S}_{con}$ for $i = 1, \ldots, n$ and $t \in \mathcal{T}_G(\Sigma, S)$.
- $\neg P$ is a formula if $Vars(P) \cap \mathcal{V}_{abs} = \emptyset$.
- $P \wedge Q$ is a formula if $Vars(P) \cap Vars(Q) = \emptyset$.
- $P \vee Q$ is a formula if $Vars(P) \cap \mathcal{V}_{abs} = Vars(Q) \cap \mathcal{V}_{abs}$ and for each variable $x \in Vars(P)$ either it occurs as a primary or secondary occurrence but not both.
- $(\exists x : S)P$ is a formula where x can be both primary and secondary occurrence in P.

where further connectives like T, F, $\Rightarrow$, $\Leftrightarrow$ and $\forall$ are defined as the standard abbreviations. $Vars(P)$ denotes the variables occurring in P. The occurrence of the variable x in a Left Hand Side (LHS) of the formula $x = t$ is called a *primary occurrence*, otherwise it is a *secondary occurrence*. Note that by our syntax definition, only abstract variables have secondary occurrences. We say a DF formula P is of type $U \rightarrow V$ iff (i) the set of abstract primary variables of P is equal to V_{abs}, (ii) the set of secondary abstract variables is a subset of U_{abs} and (iii) the concrete variables have occurrences in a set $U_{con} \cup V_{con}$. Intuitively, the set U represents the *independent variables* while V represents the *dependent variables*.

Moreover, we call such x a *dependent variable* and the variables occurring in t *independent variables*. Thus a formula P is of type $U \rightarrow V$ where U is a set of independent variables and V is a set of dependant variables. In the absence of abstract sorts, the sets of variables U and V play symmetrical roles.

3.4 Semantics

A Σ-structure $\mathcal{M}$ consists of:

- $\mathcal{D}$, a carrier set, is defined as the union of the denotations for each Sort S i.e. $\mathcal{D} = \bigcup\{\mathcal{D}_S : S \in \mathcal{S}\}$ such that if $S \in \mathcal{S}_{abs}$ then $\mathcal{D}_S$ is non-empty set and if $S \in \mathcal{S}_{con}$ then $\mathcal{D}_S = \{a_1, \ldots, a_n\}$ where $a_i \neq a_j$ for $1 \leq i < j \leq n$.
- a n-ary function $\mathcal{M}(f) : \mathcal{D}^n \rightarrow \mathcal{D}$ for every n-ary function symbol f.
- a n-ary cross-operator $\mathcal{M}(p) : \mathcal{D}^n \rightarrow \mathcal{D}$ for every n-ary cross-operator symbol p.

We say a partial mapping $\phi : \mathcal{V} \rightarrow \mathcal{D}$ is a partial Σ-assignment iff $\phi(x) \in \mathcal{D}_{\eta(x)}$ for every variable $x \in \mathsf{Dom}(\phi)$. We assume that the structure $\mathcal{M}$ is fixed and the formal definition of the semantics relative to the mapping ϕ is:

$$
\begin{array}{rcl}
[\![x]\!]^\phi & = & \phi(x) \quad \text{for } x \in \mathcal{V} \\
[\![f(t_1, \ldots, t_n)]\!]^\phi & = & \mathcal{M}(f)([\![t_1]\!]^\phi, \ldots, [\![t_n]\!]^\phi) \\
[\![x = t]\!]^\phi = tt & \text{iff} & [\![x]\!]^\phi = [\![t]\!]^\phi \\
[\![p(t_1, \ldots, t_n)]\!]^\phi = tt & \text{iff} & \mathcal{M}(p)([\![t_1]\!]^\phi, \ldots, [\![t_n]\!]^\phi) = tt \\
[\![\neg P]\!]^\phi = tt & \text{iff} & [\![P]\!]^\phi = ff \\
[\![P \wedge Q]\!]^\phi = tt & \text{iff} & [\![P]\!]^\phi = tt \text{ and } [\![Q]\!]^\phi = tt \\
[\![(\exists x : S)P]\!]^\phi = tt & \text{iff} & [\![P]\!]^{\phi[c/x]} = tt \\
 & & \text{for some } c \in \mathcal{D}_S \\
 & & \text{such that } \phi[c/x] \text{ is like } \phi \\
 & & \text{but maps } x \text{ to } c
\end{array}
$$

The remaining logical connectives are interpreted as usual.

3.5 MDG Structure

MDGs subsume the class of Bryant's (ROBDD) while accommodating abstract data and uninterpreted function symbols. An MDG of type $U \rightarrow V$ can be seen as a Directed Acyclic Graph (DAG) G with one root and ordered edges, such that:

1. Every leaf node is labeled by the formula T, except if G has a single node, which may be labeled T or F.
2. For every internal node N, either
   (a) N is labeled by $\mathcal{T}(U \cup V_{con}, \mathcal{L}_{CO}, \mathcal{S})$ and the edges that issue from N are labeled by $\mathcal{T}_G(\mathcal{S}_{con})$, or
   (b) N is labeled by a variable in V_{abs} and the edges that issue from N are labeled by $\mathcal{T}(U_{abs}, \mathcal{L}_F, \mathcal{S})$

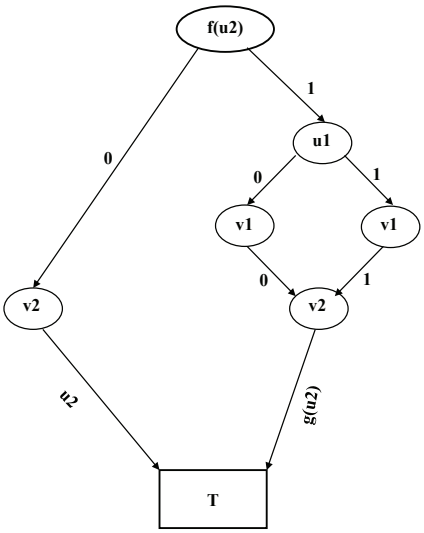
Terms are made out of sorts, constants, variables, and function symbols. Two kinds of sorts are distinguished:

- Concrete sort: is equipped with finite enumerations, lists of individual constants. They are used to represent control signals.
- Abstract sort: has no enumeration available. It uses first order terms to represent data signals.

MDGs represent and manipulate a certain subset of first order formulae, which we call Directed Formulae (DFs) and therefore must be *reduced* and *ordered* like ROBDD [6]. DFs can represent the transition and output relations of a state machine, as well as the set of possible initial states and the sets of states that arise during reachability analysis. Consequently, DFs must obey a set of well-formedness conditions given in [11]. Intuitively, these conditions represent pre-conditions for some basic MDG algorithms which are mainly disjunction, Relational Product and Pruning by Subsumption. We will investigate these algorithms in the next section.

In order to illustrate MDGs, we consider the following example DF of type $\{u_1, u_2\} \rightarrow \{v_1, v_2\}$, where u_1 and v_1 are variables of a concrete sort *bool* with enumeration $\{0, 1\}$ while u_2 and v_2 are variables of an abstract sort α, g is an abstract function symbol of type $\alpha \rightarrow \alpha$ and f is a cross-operator of type $\alpha \rightarrow bool$. Then, Figure 1 shows the MDGs representing this example as well as its corresponding DF formula.

Like for ROBDDs, a symbol order according to which an MDG is built could be provided by the user. This symbol order can affect critically the size of the generated MDG. Otherwise, MDG can use an automatic dynamic ordering.



(f(u2)=0) /\ (v2=u2)) \/
((f(u2)=1) /\ (u1=0) /\ (v1=0) /\ (v2=g(u2))) \/
((f(u2)=1) /\ (u1=1) /\ (v1=1) /\ (v2=g(u2)))

Figure 1: Example of MDG and its Corresponding DF Formula

The MDG model checking is based on an abstract implicit state enumeration. The system is expressed as an Abstract State Machine (ASM) and the properties to be verified are expressed by formulae in $\mathcal{L}_{MDG}$ [28]. $\mathcal{L}_{MDG}$ atomic formulae are Boolean constants (True and False), or equations of the form $(t_1 = t_2)$, where t_1 is an ASM variable (input, output or state variable) and t_2 is either an ASM system variable, an individual constant, an ordinary variable or a function of ordinary variables. Ordinary variables are defined to memorize the values of the system variables in the current state.

The MDG operations and verification procedures are packaged as a tool and implemented in Prolog [10]. The MDG-tool provide facilities for invariant checking, verification of combinational circuits, sequential verification, equivalence checking of two state machines and model checking.

4 MDG Construction

Let P be an MDG of the form:

$$\mathrm{MDG}(x, \{a_1, \ldots, a_m\}, \{l_1, \ldots, l_n\}, \{m_1, \ldots, m_n\})$$

then $\mathsf{top}(P)$ denotes the root node x, $\mathsf{arg}(P)$ denotes the set $\{a_1, \ldots, a_m\}$ (eventually empty) of the cross-operator arguments, $\mathsf{edges}(P)$ denotes a non-empty set $\{l_1, \ldots, l_n\}$ of labels (edges), and $\mathsf{childs}(P)$ denotes a non-empty set $\{m_1, \ldots, m_n\}$ of sub-MDGs.

In a ROBDD, Boolean variables are used to encode the enumerated types. This can be done by simply using a recursive function that divides the values into two subsets of roughly equal size, creates a variable to distinguish between them,

and then recurses on the two subsets. It results in an Algebraic Decision Diagram (ADD) [22] that extends BDD's by allowing values from arbitrary finite domain to be associated with the terminal nodes. Then this ADD is translated to ROBDD. Due to the presence of abstract sorts, this approach cannot be used for MDG. Also, in Logic with Equality and Uninterpreted Functions (LEUF), or more precisely, Quantifier-Free First-Order Logic with Equality and Uninterpreted Functions does not have universal or existential quantifiers, but has the equal sign as a special predicate. Therefore, an equation (atomic formula with equality) is used to represent directly the MDG without encoding the concrete domains. We will use the notation $Eq(x, \{a_1, \ldots, a_n\}, l)$ to denote an MDG such that (i) the root node is labeled with x and the (eventually empty) set $\{a_1, \ldots, a_n\}$ (ii) the edge is labeled with l and (iii) the terminal node is labeled with T.

4.1 Generalized-If-Then-Else (GITE)

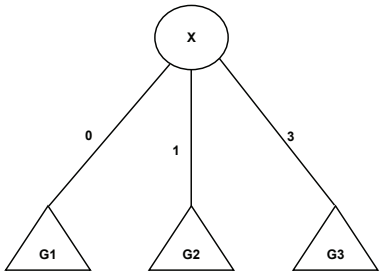
Given a ROBDD b, a boolean function f represented by b is recursively defined by:

$$f = (\neg x \wedge f_{x=0}) \vee (x \wedge f_{x=1})$$

where x is the variable in b's root node and the cofactor function $f_{x=0}$ is defined by the reachable subgraph of b's 0-branch child. Similarly, $f_{x=1}$ is recursively defined by the reachable subgraph of b's 1-branch child. Therefore a ROBDD node can be naturally represented by an If-Then-Else statement, i.e. $\mathsf{ITE}(x, f_{x=1}, f_{x=0})$.

Given a variable ordering and three ROBDDs f, g and h, the ROBDD result of f, g and h is easily constructed by Shannon's expansion in the depth-first manner. This expansion process repeats recursively following the given variable order for the Boolean variables in f, g, and h. The base case (also called the terminal case) is when f, g or h are representing a terminal node (i.e. Tor F node). For example, $\mathsf{ITE}(\mathsf{T}, g, h)$ can be trivially evaluated to g. The recursive process will terminate because restricting all the variables of functions produces constant functions T or F. At the end of the expansion phase, the uniqueness of ROBDD representation is ensured by reducing expressions like $\mathsf{ITE}(x, f, f)$ to f. This bottom-up reduction phase is performed in the reverse order of the expansion phase. Finally, since all the boolean connectives can be expressed as If-Then-Else statement, this construction provides a uniform way to build arbitrary Boolean functions.

Similarly, our goal is to provide the same construction for MDGs. The definition of the cofactor function is made upon the following observation. Assuming that x ranges over $\{0, 1, 3\}$ and that there could be, say, only three edges issuing from the root, as in the following graph:



where G_1, G_2 and G_3 represent the formulae P_1, P_2 and P_3 respectively, then this MDG could represent the formula

$$(x = 0 \wedge P_1) \vee (x = 1 \wedge P_2) \vee (x = 3 \wedge P_3)$$

When x denotes 2, this formula is simply a false sentence. Therefore, the cofactor $P_{x=l,\mathsf{arg}(x)}$ with respect to a (concrete or abstract) variable x restricted to label l and a set of the cross-operator arguments $\mathsf{arg}(x)$ (possibly empty) is defined as:

$$P_{x=l,\mathsf{arg}(x)} = \begin{cases} P & \text{if } x < \mathsf{top}(P) \\ m_i & \text{if } \exists i (l = l_i) \wedge (\mathsf{arg}(P) = \mathsf{arg}(x)) \\ \mathsf{F} & \text{otherwise} \end{cases}$$

While concrete sorts have enumerations, abstract sorts do not. To overcome this problem, we can collect all the labels of the abstract variable x from the MDGs involved in the construction. This task is achieved by the function enum which is defined as:

$$\mathsf{enum}(x, P) = \begin{cases} S_{con} & \text{if } x \in S_{con} \text{ and } \mathsf{top}(P) = x \\ \mathsf{edges}(P) & \text{if } x \in S_{abs} \text{ and } \mathsf{top}(P) = x \end{cases}$$

This function exploits the variable ordering, hence there is no need to traverse all the children of P to collect the edges. Moreover, we assume that the set of edges are ordered.

Our GITE algorithm takes as input three MDGs P, Q and H of type $U_i \rightarrow V_i$ for $i = 1..3$ respectively and produces an MDG $R = GITE(P, Q, H)$ of type $\bigcup_{1 \leq i \leq 3} U_i \rightarrow \bigcup_{1 \leq i \leq 3} V_i$ such that $\models R \Leftrightarrow (P \wedge Q) \vee (\neg P \wedge H)$. Such MDG R does not always exist due to abstract variables. For example, let x be an abstract variable and a be an abstract generic constant. Let P be $x = a$ (i.e., an MDG with a root node labeled x and a single edge labeled a leading to T), then there is no MDG representing the formula $\neg(x = a)$. Thus there can be no algorithm for *general negation*. On the other hand, it is easy to compute a formula logically equivalent to $\neg P$ that has no nodes labeled by abstract variables. Similarly, there does not always exist an MDG R such that $\models R \Leftrightarrow (P \vee Q)$. For example, let x and y be distinct abstract variables, and a and b distinct abstract generic constants, then there exists no well-formed MDG representing the formula $x = a \vee y = b$. Finally, it may be impossible to compute the conjunction

of two MDGs whose root nodes have the same label, if that label is an abstract variable (i.e., $x = a \wedge x = b$). Note all these formulae are not DFs since they do not respect the syntax constraints defined in Section 3. Moreover, we claim that the logical equivalence between R and $(P \wedge Q) \vee (\neg P \wedge H)$ can be shown independent of the negation of P, particularly when the top symbol of P is an abstract variable. For example, it is easy to show that $\models (x = a \vee x = b) \Leftrightarrow (x = a \wedge \mathsf{T}) \vee (\neg(x = a) \wedge x = b)$ in classical logic. The detailed algorithm is given below:

$GITE(P, Q, H)$

1. if (terminal case) then
2. return (R = trivial result);
3. else
4. if (computed table has entry $\{(P, Q, H), R\}$) then
5. return R from computed table ;
6. else
7. x = top variable of P, Q and H;
8. $S = \mathsf{enum}(x, P, Q, H)$;
9. $a = \mathsf{arg}(x)$;
10. $l, m = \emptyset$;
11. for (each s s.t. $s \in S$) do
12. $R = GITE(P_{x=s,a}, Q_{x=s,a}, H_{x=s,a})$;
13. if ($R \neq \mathsf{F}$) then
14. append(l,s); append(m,R);
15. endif
16. endfor
17. if($l = \emptyset$) then ($R = \mathsf{F}$);
18. else R = find_or_add_unique(x, a, l, m);
19. endif
20. insert (P, Q, H, R) in the computed table
21. return R;
22. endif
23. endif

The resulting MDG is constructed by recursively performing Shannon's expansion. This recursive expansion ends when a terminal node is reached (lines 1 and 2) or when it is found in the computed table (line 4 and 5). A computed table stores previously computed results to avoid repeating work that was done previously. Line 7 determines the top variable of P, Q and H. Line 8 extracts a set of labels (edges) S according to the top variable sort. When this sort is concrete, then S is equal to the enumeration of this sort. Otherwise, we collect the labels from the MDGs involved in the construction. Line 9 and 10 extract eventually the arguments if the top variable is a cross-operator and initialize the new set of labels and MDGs to be constructed. Lines 11 to 16 recursively perform Shannon's expansion on the cofactor in respect to S and computes the new edges and MDGs by discarding the elements of S that result in a terminal MDG F. At the end of the expansion (line 17), either the resulting MDG is F or the reduction step and uniqueness of the resulting MDG are performed (line 18). The reduction step is applied only on the concrete sorts. Therefore a node is redundant if all the edges are in the enumeration of the concrete sort and the corresponding MDGs are the same.

To prove the termination of a recursive call, we have to prove that an infinite sequence of recursive calls does not exist i.e. the loop body executes a finite number of times. We have to define a mapping function $v = depth(P) + depth(Q)+depth(H)$ which represents the depth or size of MDGs P, Q and H. Where the depth function represents the number of nodes in the longest path of an MDG. It is clear from the definition of the cofactor that v is decreasing after each call of $GITE$ and since the MDGs P, Q and H are finite then the termination is guaranteed.

The correctness procedure consists of applying the GITE algorithm over P, Q and H MDGs and get the result R as an MDG. Then using *FormulaMDG* algorithm shown below, we build its corresponding formula and compare it with the formula obtained by applying the GITE algorithm over P, Q and H.

$FormulaMDG(P)$

```
1.  if top(P) = 0 then
2.      return F;
3.  else if top(P) = 1 then
4.      return T;
5.  else
6.      x=top(P);
7.      S=enum(x,P);
```

8. a=arg(x);
9. for (each s s.t. $s \in S$) do
10. DF= $\vee_{s \in S}(x = s) \wedge FormulaMDG(P_{x=s,a})$;
11. endfor
12. return DF;
13. endif

To keep the formula resulted from the *FormulaMDG* algorithm in DF format (disjunction of conjunction of equations) we add a distribution rule which allows distribution of conjunction over the disjunction such that: $x \wedge (y \vee z) \Leftrightarrow (x \wedge y) \vee (x \wedge z)$.

Theorem 4.1. *The GITE algorithm is correct and terminates*

PROOF SKETCH: By induction on P. The MDG resulted from the GITE algorithm is feeded to the *FormulaMDG* to get its corresponding DF and then compared with the *ite-operator* result. The correctness criteria for the proof of GITE algorithm is shown in the following:

if $R = GITE(P, Q, H)$ *then* $FormulaMDG(R) \equiv (P \wedge Q) \vee (\neg P \wedge H)$

ASSUME: 1. P, Q and H are finite MDGs and represent a well-formed DF.

PROVE: True

$< 1 >$ 1. ***CASE:*** Induction on P:

PROOF:

1. The Base case:
   - When P represents a terminal node which may be labeled as T or F, the result R of the GITE algorithm will be either Q or H, respectively.
   - When the entry of the call memory function for the GITE of P, Q and H in the computed table is computed in the hash table so the function return the value of R from the computed table (follow the uniqueness condition) and terminate.

   The *FormulaMDG* will return the corresponding formula for the MDG R. This result is equivalent to the one resulted from the *ite-operator* algorithm (trivial case).
2. The Induction case: P could be one of the below cases:
   (a) $x = t$ if x is a concrete variable.
   (b) $f(t1, \cdots, tn) = t$ if f is a cross-operator.
   (c) $x = t$ if x is an abstract variable.
   (d) $P1 \wedge P2$.

(e) $P1 \vee P2$.

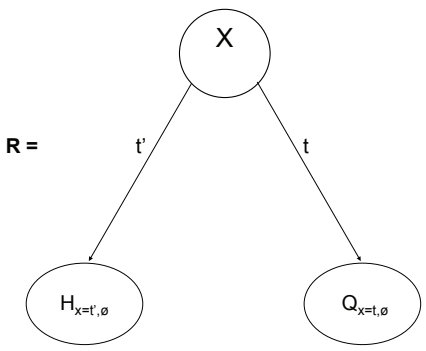
(f) $\neg P$.

(g) $(\exists x : S)P$.

The most difficult case when $P = (x = t)$, where $x \in Xabs$ is the top variable of P. We show this case in details while the other cases are straightforward. Applying the GITE algorithm on P, Q and H results: $Q_{x=t,\phi}$ in the case of $(x = t)$ and in the negation case where $\neg(x = t)$, we generate a unique fresh variable t' from the set of secondary variables (independent variables), and thus we have $\neg(x = t) = (x = t')$ such that $t \neq t'$ and the result will be $H_{x=t',\phi}$.

Then it is easy to extract a DF from the above MDG using the *FormulaMDG* algorithm such that: $R = ((x = t) \wedge Q_{x=t,\phi}) \vee ((x = t') \wedge H_{x=t',\phi})$.

This formula is the same resulted from applying the *ite-operator* such that: $(P \wedge Q) \vee (\neg P \wedge H) = ((x = t) \wedge Q) \vee (\neg(x = t) \wedge H)$. An MDG sketch representing the formula $GITE((x = t), Q, H)$ is shown below:



To prove the correctness by induction on P, we have two cases:

- When $(x = t)$ then $FormulaMDG(Q_{x=t,\phi}) = Q$.
- When $\neg(x = t) = (x = t')$ then $FormulaMDG(H_{x=t',\phi}) = H$.

Which is equivalent to the result from the GITE algorithm.

$< 1 > 2$. Q. E. D.

PROOF: Step $< 1 > 1$ and assumption 1.

Lets take a simple example for illustration purposes, if $P1 = (x = a)$ and $P2 = (y = b)$, where x is the top of $P1 \wedge P2$, then applying the ITE algorithm $ITE(x = a \wedge y = b, Q, H)$ results,
x is the top variable, $S = \{a\}$ and $a = \phi$ then entering the first loop will result
$ITE(x = a, Qx = a, Hx = a)$, and again y is the top variable, $S = \{b\}$ and $a = \phi$ then entering the second loop will result
$ITE(T, (Qx = a)y = b, (Hx = a)y = b)$.

Then its easy to extract a DF from the above MDG as shown below:
$R = [(x = a) \wedge (y = b) \wedge Qx = a] \vee$
$[\neg(x = a) \wedge H] \vee$
$[(x = a) \wedge \neg(y = b) \wedge Hx = a]$

This formula is the same one resulted from applying the *ite-operator* and hence they are equivalent. An MDG sketch is shown in Figure 2.

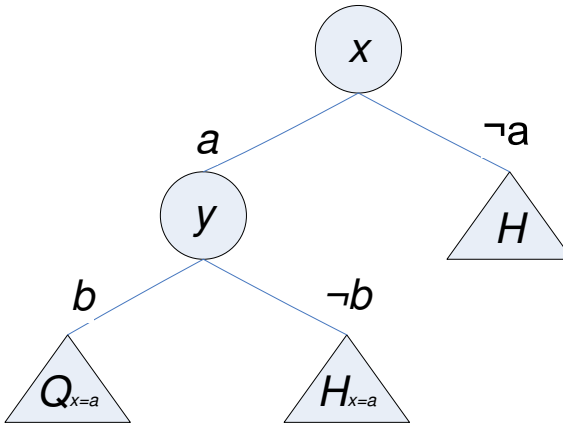


Figure 2: $ITE((P1 \wedge P2), Q, H)$

4.2 Relational Product (RelP)

The Relational Product combines conjunction and existential quantification in one step. We provide an algorithm that extends the ROBDD relational product. It takes the conjunction of two MDGs having disjoint sets of abstract primary variables and existentially quantifies with respect to some abstract or concrete variables that have primary occurrence in at least one of the MDGs. The primary occurrence of an abstract variable in one MDG can be a secondary occurrence in the other MDG.

For this reason, we have introduced a substitution that includes those variables during the construction (i.e., the secondary variables are implicitly quantified). The substitution is applied in the reverse order of the expansion phase on the edges labeled with secondary occurrence variables and cross-operators arguments. However, while the ordering of variables cannot be preserved in case of cross-operators, there may exist redundant or contradictory MDG result during intermediate steps.

For example, let $x < m < M$ be an ordering of variables and let P be $leq(x, m) = 1 \wedge leq(x, M) = 0$ where x, m and M are secondary abstract variables that having primary occurrences in another MDG, say, Q, and $\sigma = \{x \mapsto x\#3, m \mapsto x\#2, M \mapsto x\#1\}$, then the resulting MDG $leq(x\#3, x\#2) = 1 \wedge leq(x\#3, x\#1)) = 0$ does not preserve the order the variable $x\#i$ serves as a symbolic value of x at the i^{th} step and $i < j \Rightarrow x\#i < x\#j$. Therefore, we will distinguish the case of the cross-operator and provide a special construction for it.

Let E be the set of quantified variables, our algorithm takes two MDGs P, Q of type $U_i \rightarrow V_i$ for $i = 1..2$ and a substitution σ with $\mathsf{Dom}(\sigma) = E$ and returns an MDG $R = \mathsf{RelP}(P, Q, E, \sigma)$ of type $(\bigcup_{1 \leq i \leq 2} U_i \setminus \bigcup_{1 \leq i \leq 2} V_i) \rightarrow (\bigcup_{1 \leq i \leq 2} V_i \setminus \bigcup_{1 \leq i \leq 2} U_i)$ such that $\models R \Leftrightarrow \exists E(P \wedge Q)$.

$\mathsf{RelP}(P, Q, E, \sigma)$

1. if (terminal case) then
2. return (R = trivial result);
3. else
4. if (computed table has entry $\{(P, Q, E, \sigma), R\}$) then
5. return R from computed table ;
6. else
7. x = top variable of P, Q
8. $S = \mathsf{enum}(x, P, Q)$;
9. $a = \mathsf{arg}(x)$;
10. $l, m = \emptyset$;
11. for (each s s.t. $s \in S$) do
12. $R = \mathsf{RelP}(P_{x=s,a}, Q_{x=s,a}, E, Extend(\sigma, x, s, E))$;
13. if ($R \neq \mathsf{F}$) then
14. append(l,s); append(m,R);
15. endif
16. endfor
17. if($l = \emptyset$) then ($R = \mathsf{F}$);
18. else
19. if($x \in E$) then
20. $R = Or(m)$
21. else
22. if($a = \emptyset$) then
23. R = find_or_add_unique$(x, a, \sigma(l), m)$;
24. else
25. $R = \mathsf{F}$
26. for (each $l_i \in l$ and $m_i \in m$)
27. $R = Or(R, And(Eq(x, \sigma(a), l_i), m_i))$
28. endfor
29. endif
30. endif
31. endif

32. insert (P, Q, E, σ, R) in the computed table
33. return R
34. endif
35. endif

Like ROBDD Relational Product algorithm, RelP uses a result cache. If the entry (P, Q, E, σ) is in the cache, then it means that a previous call to $\mathsf{RelP}(P, Q, E, \sigma)$ returned R as result. Lines 7 to 16 apply recursively the Relational Product with respect to a top symbol x where $Extend(\sigma, x, s, E)$ returns $\sigma \oplus \{s/x\}$ if $x \in E$ otherwise it returns σ. Lines 19 to 31 apply either quantification or conjunction depending whether the variable x occurs in E or not. As explained above, we distinguish the cross-operators case (lines 25 to 28), where we construct a new MDG that respects the ordering of variables, thus avoiding any contradictions.

Theorem 4.2. *The* RelP *algorithm is correct and terminates*

PROOF SKETCH: By induction on P. The MDG resulted from the RelP algorithm is feeded to the *FormulaMDG* to get its corresponding DF and then compared with the result of $\exists E(P \wedge Q)$. The correctness criteria for the proof of RelP algorithm is shown in the following:

if $R = RelP(P, Q, E, \sigma)$ *then* $FormulaMDG(R) \equiv \exists E(P \wedge Q)$

ASSUME: 1. P and Q are finite MDGs and represent a well-formed DF.

PROVE: True

$< 1 > 1$. *CASE:* Induction on P:

PROOF:

1. The Base case:
   - When P represents a terminal node which may be labeled as T or F, the result R of the RelP algorithm will be either $\exists E(Q)$ or F, respectively.
   - When the entry of the call memory function for the RelP of P and Q in the computed table is computed in the hash table so the function return the value of R from the computed table (follow the uniqueness condition) and terminate.

   The *FormulaMDG* will return the corresponding formula for the MDG R. This result is equivalent to the one resulted from the $\exists E(P \wedge Q)$ (trivial case).

2. The Induction case: P could be one of the seven cases mentioned in the proof of Theorem 4.1. Then, we construct a DF for

the result obtained from the RelP algorithm using the *FormulaMDG* and compare it with the result from the formula $\exists E(P \wedge Q)$ and hence they are equivalent.

$< 1 > 2$. Q. E. D.

PROOF: Step $< 1 > 1$ and assumption 1.

For the case when $P = (x = t)$, where $x \in Xabs$ is the top variable of P. P and Q must have the same set of abstract variables. Applying the RelP algorithm on P, Q and E results:

x is the top variable, $S = \{t\}$ and $a = \phi$ then entering the first loop will result $RelP(x = t, Q, E, \sigma)$, then entering the second loop will result after applying the substitution and existentially quantifies over the variables E

$RelP(T, Q_{x=t,\phi}, E, \{\sigma \oplus \{s/x\}\}) = Q_{x=t,\phi}$.

Then its easy to construct a DF from the above MDG using the *FormulaMDG* as: $R = ((x = t) \wedge Q_{x=t,\phi}))$. This formula is the same obtained from $\exists E(P \wedge Q)$.

4.3 Pruning by Subsumption (PbyS)

The Pruning by Subsumption algorithm approximates the difference of sets represented by MDGs (i.e. DFs). We propose a new algorithm which uses restricted operators and builds an MDG in a similar manner as GITE does. The proposed algorithm improves the original one in many ways. First, the expansion is done only on the first argument i.e., P rather than on P and Q. Indeed, we can view each disjunct of DF as a state description. Without loss of generality, we can assume that P and Q contain only one disjunct. Then, we can say that P is subsumed by Q if and only if there exists a substitution σ such that the state description of $Q\sigma$ is a subset of the state description of P. Therefore the size of P should be at least equal to the size of Q. Next, when the top variable of Q is less than the top variable of P, it is obvious that the state description of Q is not a subset of P. Hence, the cofactor of Q should be F, which improves drastically the original algorithm. Finally, when P and Q have the same top symbol cross-operator but there is a mismatch either on the edges or on the arguments, the cofactor of Q is Q itself and we discard the substitution if any resulting from the unification of their arguments. These observations lead to a new restricted operator defined as follows.

Given an MDG Q, the restriction of Q with respect to a variable x, an edge l, a set of cross-operator arguments $\mathsf{arg}(x)$ and a substitution σ, written $Q|_{x=l,\mathsf{arg}(x),\sigma}$, returns a pair of MDG-substitution $\langle m, \sigma' \rangle$ as:

$$Q|_{x=l,\mathsf{arg}(x),\sigma} = \begin{cases} \langle Q,\sigma\rangle & \text{if } x < \mathsf{top}(Q) \\ \langle \mathsf{F}\ ,\sigma\rangle & \text{if } \mathsf{top}(Q) < x \\ \langle m_i,\sigma'\rangle & \text{if } (\exists i)(l = l_i\sigma') \wedge \mathsf{arg}(Q) = \mathsf{arg}(x) = \emptyset \\ \langle Q,\sigma\rangle & \text{if } (\neg\exists i)(l = l_i\sigma') \wedge \mathsf{arg}(Q) = \mathsf{arg}(x) = \emptyset \\ \langle m_i,\sigma''\rangle & \text{if } \exists i(l = l_i\sigma'') \wedge (\mathsf{arg}(Q)\sigma'' = \mathsf{arg}(x)) \\ \langle Q,\sigma\rangle & \text{if } \neg\exists i(l = l_i\sigma'') \vee (\mathsf{arg}(Q)\sigma'' \neq \mathsf{arg}(x)) \\ \langle \mathsf{F}\ ,\sigma\rangle & \text{otherwise} \end{cases}$$

where $\sigma' = \sigma \oplus \{l_i \mapsto l\}$ and $\sigma'' = \sigma \oplus \{\mathsf{arg}(Q) \mapsto \mathsf{arg}(x)\}$.

Our PbyS algorithm takes two MDGs P and Q of type $U \to V_1$ and $U \to V_2$ and a substitution σ initially equal to the identity and produces an MDG P' of type $U \to V_1$ such that P' is derivable from P by *pruning* some paths such that $\models P \vee (\exists U)Q \equiv P' \vee (\exists U)Q$. The paths that are removed from P are subsumed by Q, hence the name of the algorithm. If $P' = \mathsf{F}$ then, we can view P' as a logical difference of P and $(\exists U)Q$ i.e. $\models P \Rightarrow (\exists U)Q$. The detailed algorithm is given below:

PbyS(P, Q, σ)

1. if (terminal case) then return (P' = trivial result);
2. else if (PbyS table has entry $\{(P, Q, \sigma), P'\}$) then
3. return P' from PbyS table ;
4. else
5. $x = \mathsf{top}(P)$; $l, m = \emptyset$; $a = \mathsf{arg}(P)$;
6. for (each $s \in \mathsf{edges}(P)$) do
7. $P' = P_{x=s,a}$;
8. stack $= Q|_{x=s,a,\sigma}$;
9. while stack is not empty;
10. $\langle m', \sigma'\rangle$= pop stack;
11. $P' = \mathsf{PbyS}(P', m', \sigma')$;
12. if ($P' = \mathsf{F}$) break;
13. endwhile;
14. if($P' \neq \mathsf{F}$) then
15. append(l,s); append(m,P');
16. endif

17. endfor;
18. if($l = \emptyset$) then ($P' = \mathsf{F}$);
19. else $P' =$ find_or_add_unique(x, a, l, m);
20. update PbyS table ($\{(P, Q, \sigma), P'\}$) ;
21. return P';
22. endif

The result MDG is constructed by recursively performing the restricted operators introduced on P and Q until a terminal node is reached (line 1) or when it is found in the PbyS table (line 2). Line 5 determines the top variable of P and the cross-operator arguments (if possible) and initializes the new edges and children to be constructed. Then from each edge issuing from the node x (line 6), we extract the cofactors of P and Q where the cofactors of Q are pairs of MDG-substitution stored in a stack. Lines 9 to 13 check whether the cofactors of P, written P', is subsumed by one of the Q paths. If so (line 12) then there is no need to try the other cofactors of Q and therefore we continue with the remaining cofactors of P and we discard P'. Otherwise, the edge and this cofactor are added to the corresponding table (lines 14-16). When we have processed all the cofactors of P (line 18) either all the paths of P are subsumed by P and thus the result MDG is F, or the reduction step and uniqueness of the resulting MDG are performed (line 20) with all or some paths of P that not subsumed.

Theorem 4.3. *The* PbyS *algorithm is correct and terminates*

PROOF SKETCH: By induction on P. The MDG resulted from the PbyS algorithm is feeded to the *FormulaMDG* to get its corresponding DF and then compared with the result of $P \vee (\exists U)Q \equiv P' \vee (\exists U)Q$. The correctness criteria for the proof of PbyS algorithm is shown in the following:

if $P\prime = PbyS(P, Q, \sigma)$ *then* $FormulaMDG(P\prime) \equiv P \vee (\exists U)Q \equiv P' \vee (\exists U)Q$

ASSUME: 1. P and Q are finite MDGs and represent a well-formed DF

PROVE: True

$< 1 > 1$. *CASE:* Induction on P:

PROOF:

1. The Base case:
   - When P represents a terminal node which may be labeled as T or F, the result P' of the PbyS algorithm will be either T or F, respectively.
   - When the entry of the call memory function for the PbyS of P and Q in

the computed table is computed in the hash table so the function return the value of P' from the computed table (follow the uniqueness condition) and terminate.

The *FormulaMDG* will return the corresponding formula for the MDG P'. This result is equivalent to the one resulted from the $P \vee (\exists U)Q \equiv P' \vee (\exists U)Q$ (trivial case).

2. The Induction case: P could be one of the seven cases mentioned in the proof of Theorem 4.1. Then, we construct a DF for the result obtained from the PbyS algorithm using the *FormulaMDG* and compare it with the result from the formula $P' \vee (\exists U)Q$ and hence $\models P \vee (\exists U)Q \equiv P' \vee (\exists U)Q$ and hence they are equivalent.

$< 1 > 2$. Q. E. D.

PROOF: Step $< 1 > 1$ and assumption 1.

For example, take the case when $P = (x = t)$, where $x \in Xabs$ is the top variable of P. P and Q must have the same set of abstract variables. Applying the RelP algorithm on P and Q results:
x is the top variable, $S = \{t\}$, $\sigma = \{\}$ and $a = \phi$ then entering the first loop will result $PbyS(x = t, Q, \sigma)$, then entering the second loop will result $PbyS(T, Q_{x=t,\phi}, \sigma) = Q_{x=t,\phi}$.

Then its easy to construct a DF from the above MDG using the *FormulaMDG* as: $P' = Q_{x=t,\phi}$. This formula is the same obtained from $P \vee (\exists U)Q$ and hence $\models P \vee (\exists U)Q \equiv P' \vee (\exists U)Q$.

5 NuMDG Tool

5.1 Overview

A high level description of NuMDG is given in Figure 3. In the future, we will provide an open source tool with many functionalities independent of the model checking engine used. Like NuSMV [9], the tool will be able to process files written in an extension of the SMV language with abstract sort and uninterpreted functions. In this language, finite state machines are described by using instantiation mechanism of modules and processes, corresponding to synchronous and asynchronous composition respectively. The requirements are written in CTL, LTL or in a first-order subset of temporal logic.

An (extended) SMV file is processed in several phases. The first phase analyzes the input file with different layers in order to construct an internal representation of the model. The construction starts from modular description of a model M and of a set of properties $P_1, \ldots P_n$. The flattening step consists of eliminating modules and processes and producing a

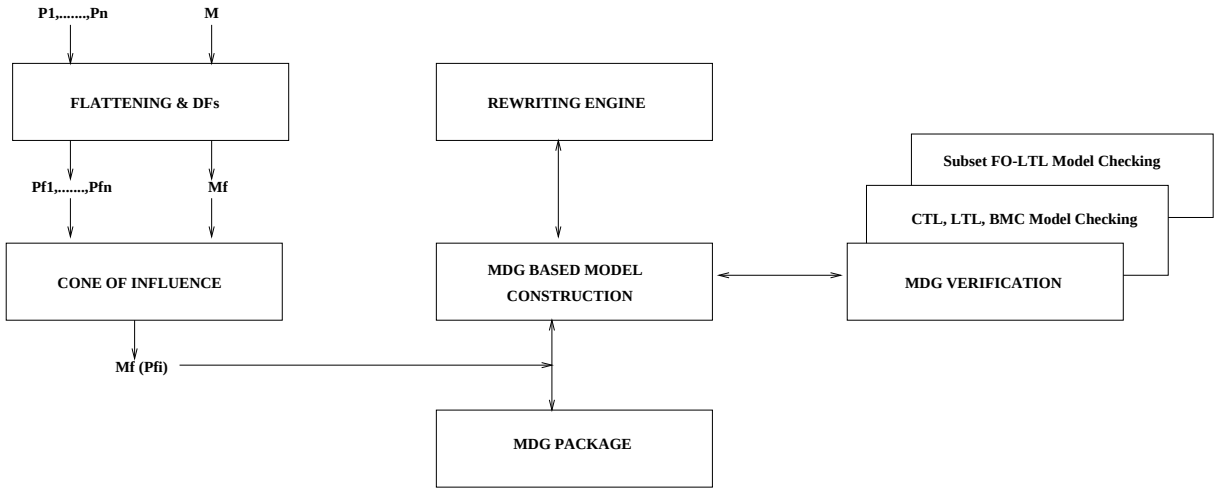


Figure 3: Internal structure of NuMDG

synchronous flat model, where each variable is given an absolute name. The second step, called DF, maps each expression in the flat model to a directed formula, thus obtaining the corresponding flattened directed model M_f. Compared to SMV-based tools, there is no boolean encoding. Hence, some interpreted predicates and arithmetic functions are not supported in a straightforward manner. The same reduction is applied to the properties P_i, thus obtaining the corresponding flattened directed formula P_{fi}. By cone of influence, we restrict the analysis of each property to the relevant parts of the model $M_f(P_{fi})$.

After the preprocessing phase, the user can choose the model checking engine to be used for verification. The choice is restricted by the nature of the model being described i.e. whether it supports abstract sorts and uninterpreted functions or not. In the absence of the latter, NuMDG is acting like NuSMV and should provide the same facilities including MDG-based, SAT-based model checking and different partitioning methods. For the time being, MDG-based verification includes reachability analysis and fair CTL model checking.

The rewriting engine is used during the MDG-verification if necessary when the reachability analysis does not terminate due to the presence of abstract sort and uninterpreted functions. In this case we can interpret partially some functions or predicates in order to cope with this non termination [3]. The input language supports a rewriting layer which is extracted and feeded to the rewriting engine.

5.2 Experimental results

We consider some cases from the SMV benchmark suites as benchmarks in order to measure the performance of our tool. Our objective is to build a robust and flexible symbolic model checker that accepts SMV input and providing at the same time a better mechanism for abstraction through abstract sorts and uninter-

preted functions. We have already implemented a prototype and presented below some experimental results based on some SMV benchmarks.

The first set consists of comparing NuMDG against SMV and NuSMV in respect to the number of BDD/MDG nodes allocated and to the number of BDD/MDG nodes representing the transition relation as shown in Table 1.

The table shows that the size of the MDG transition relation is much smaller. This is due to the absence of boolean encoding, i.e. we don't encode the values of model variables. However, the number of MDG allocated nodes tends to be greater. Consequently, these small (intermediate) MDGs have a negative impact on computation time and memory as illustrated by Table 2 ("-" means did not terminate).

Table 1: No. of BDD/MDG nodes comparison

Example	SMV		NuSMV		NuMDG	
	#Alloc.	# Trans.	#Alloc.	#Trans.	#Alloc.	#Trans.
Semaphore	233	69	854	67	418	53
Mutex (sync)	179	31	350	29	178	21
Mutex (async)	259	56	1625	54	701	37
Gigamax	11178	1246	81563	1242	19084	975
abp4	13884	1611	27805	1609	25507	1320

Table 2: CPU and memory comparison

Example	SMV		NuMDG	
	CPU (s)	Memory (MB)	CPU(s)	Memory (MB)
Semaphore	0.01	1.19	0.02	1.37
Mutex (async)	0.02	1.25	0.05	1.68
Gigamax	0.17	1.25	0.64	2.66
abp4	0.2	1.25	1.123	4.04
abstract abp	-	-	0.07	1.49

Compared with SMV, our NuMDG consumes more resources for the verification of the first four benchmarks. This is due to the negative impact of the intermediate MDGs during the course of computation. Including the computed cache and garbage collector frequency will absolutely help to avoid these negative impacts and hence improve the performance.

On the other hand, in the last row, we have used an abstract version of an alternating bit protocol where the bus of 16 bits is replaced by an abstract sort. The result obtained improves drastically the previous one. As a future work, we need to study the performances of computed cache and garbage collector frequency to avoid the negative impact of the intermediate MDGs during the course of computation.

6 Conclusion and Future work

We have described the basic MDG algorithms that incorporated many optimizations that will yield further improvements in the performance of MDG package. The efficiency is achieved through the use of the generalization of the If-Then-Else (ITE) operator defined in the BDD package. Consequently, we have redefined the main algorithms on which the MDG verification techniques are based, i.e, Relational Product and Pruning by Subsumption. These new algorithms descriptions are based mainly on the ROBDD ones and lifted to the realm of abstract sorts and uninterpreted functions. We have also provided the correctness proof for those algorithms the internal architecture of NuMDG.

Moreover, we have presented the internal architecture of the NuMDG tool and some experimental results based on some SMV benchmarks. From these experiments, we have identified a number of open issues and future work directions. For example, we have confirmed that NuMDG can be used to check SMV specifications. Combined with abstract sorts and uninterpreted functions, NuMDG will provide at least the same performances. However, we believe that there are many optimizations that will yield further improvements in the performance of NuMDG tool such as the effect of cache and the garbage collection should be characterized according to a rigorous evaluation methodology. We also need to perform more performance analysis through the verification of several case studies.

Challenges and Limitations

One limitation of MDG based approach is that the reachability analysis algorithm may not terminate [11] under certain circumstances due to the abstract representation of data and the "uninterpreted" nature of function symbols. This can be a severe limitation on the use of MDGs as a verification tool. For example,

consider an abstract description of a conventional (non-pipelined) micro-processor where a state variable *pc* of abstract sort represents the program counter, a generic constant *zero* of the same abstract sort denotes the initial value of *pc*, and an abstract function symbol *inc* describes how the program counter is incremented by a non-branch instruction. The MDG representing the set of reachable states of the micro-processor would contain states of the form $(pc, inc(\ldots, inc(zero), \ldots))$. Consequently, there is no finite MDG representing the set of reachable states, and hence the reachability algorithm will not terminate.

The non-termination problem was first addressed in [30], where a method based on the generalization of the state variable that causes divergence. This technique is applicable only to $processor - likeloop$ circuit and if the entrance of the loop does not start in the initial state then this generalization approach may not work.

In [2], Ait-Mohamed et al. presented an approach to dealing with the non-termination problem based on retiming and circuit transformations. Yet this technique can only be applied to specific circuit structures and can not provide a general solution to the non-termination problem. An alternative way to overcome this problem is to introduce the bounded model checking technique.

Later in [3] Ait-Mohamed et al. proposed a novel approach based on the *schematization* using ρ-term [8] to solving the non-termination problem when the generated set of states, even infinite, represents a structured domain where states share certain repetitive patterns. In general, it is not always possible to find the ρ-term which will be used in this generalization.

We are currently exploring and applying the above techniques that can mitigate this problem and that they are particularly useful in reachability analysis. Future work will also include the study of the applicability of these techniques to the reachability analysis in real designs.

References

[1] W. Ackermann. *Solvable Cases of the Decision Problem.* North-Holland Pub. Co., 1954.

[2] O. Ait-Mohamed, E. Cerny, and X. Song. MDG-based verification by retiming and combinational transformations. In *Proceedings of the 8th Great Lakes Symposium on VLSI*, pages 356–361, Lafayette, LA, USA, February 1998.

[3] O. Ait-Mohamed, X. Song, and E. Cerny. On the non-termination of MDG-based abstract state enumeration. *Theoretical Computer Science*, 300:161–179, 2003.

[4] S. Berezin, A. Biere, E. Clarke, and Y. Zhu. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In *FMCAD '98: Proceedings of the Second In-*

ternational Conference on Formal Methods in Computer-Aided Design, pages 369–386, London, UK, 1998. Springer-Verlag.

[5] R. Bryant, S. German, and M. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Trans. Comput. Logic*, 2(1):93–134, 2001.

[6] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[7] Randal E. Bryant, Steven M. German, and Miroslav N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification*, pages 470–482, London, UK, 1999. Springer-Verlag.

[8] H. Chen and J. Hsiang. Recurrence domains: their unification and application to logic programming. *Inform. and Comput.*, 122:45–69, 1995.

[9] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 359–364, London, UK, 2002. Springer-Verlag.

[10] W. Clocksin and C. Mellish. *Programming in Prolog*. Springer Verlag, 1987.

[11] F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny. Multiway decision graphs for automated hardware verification. *Formal Methods in System Design*, 10(1):7–46, February 1997.

[12] W. Damm, A. Pnueli, and S. Ruah. Herbrand automata for hardware verification. In *CONCUR '98: Proceedings of the 9th International Conference on Concurrency Theory*, pages 67–83, London, UK, 1998. Springer-Verlag.

[13] Pascal Fontaine and E. Pascal Gribomont. Using bdds with combinations of theories. In *Proceedings of the 9th International Conference on Logic for Programming and Automated Reasoning (LPAR), volume 2514 of LNCS*, pages 190–201. Springer, 2002.

[14] A. Goel, K. Sajid, H. Zhou, A. Aziz, and V. Singhal. BDD based procedures for a theory of equality with uninterpreted functions. *Form. Methods Syst. Des.*, 22(3):205–224, 2003.

[15] R. Hojati, D. L. Dill, and R. K. Brayton. Verifying linear temporal properties of data insensitive controllers using finite in-

stantiations. In *CHDL'97: Proceedings of the IFIP TC10 WG10.5 international conference on Hardware description languages and their applications : specification, modelling, verification and synthesis of microelectronic systems*, pages 60–73, London, UK, UK, 1997. Chapman & Hall, Ltd.

[16] Burch J. R. and Dill D. L. Automatic verification of pipelined microprocessor control. In D. L. Dill, editor, *Proc. Work. on Computer-Aided Verification*, number 818 in Lecture Notes in Computer Science, pages 68–80. Springer Verlag, 1994.

[17] Robert P. Kurshan, Vladdimir Levin, Marius Minea, Doron Peled, and Hüsnü Yenigün. Static partial order reduction. In *TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 345–357, London, UK, 1998. Springer-Verlag.

[18] Shuvendu K. Lahiri, Randal E. Bryant, Al E. Bryant, Amit Goel, and Muralidhar Talupur. Revisiting positive equality. In *Tools and Algorithms for the Construction and Analysis of Systems, volume 2988 of LNCS*, pages 1–15. Springer-Verlag, 2004.

[19] Y. Mokhtari, Sa'ed Abed, O. Ait Mohamed, S. Tahar, and X. Song. A new approach for the construction of multiway decision graphs. In *Proceedings of the 5th international colloquium on Theoretical Aspects of Computing*, pages 228–242, Berlin, Heidelberg, 2008. Springer-Verlag.

[20] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, pages 377–390, London, UK, 1994. Springer-Verlag.

[21] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small domains instantiations. In *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification*, pages 455–469, London, UK, 1999. Springer-Verlag.

[22] R. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic Decision Diagrams and Their Applications. In *IEEE /ACM International Conference on CAD*, pages 188–191, Santa Clara, California, 1993. IEEE Computer Society Press.

[23] Yoav Rodeh and Ofer Shtrichman. Finite instantiations in equivalence logic with uninterpreted functions. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 144–154, London, UK, 2001. Springer-Verlag.

[24] S. Tahar, X. Song, E. Cerny, , Z. Zhou, M. Langevin, and O. Ait Mohamed. Modelling and automatic verification of the fairisle ATM switch fabric using mdgs. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 18(17):955 – 972, July 1999.

[25] M. Velev. Using rewriting rules and positive equality to formally verify wide-issue out-of-order microprocessors with a reorder buffer. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 28, Washington, DC, USA, 2002. IEEE Computer Society.

[26] M. Velev. Using automatic case splits and efficient cnf translation to guide a sat-solver when formally verifying out-of-order processors. In *Artificial Intelligence and Mathematics (AI&MATH)*, pages 242–254, Fort Lauderdale, Florida, USA, 2004.

[27] Miroslav N. Velev and Randal E. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors. *J. Symb. Comput.*, 35(2):73–106, 2003.

[28] Y. Xu, X. Song, E. Cerny, and O. Ait Mohamed. Model checking for a first-order temporal logic using multiway decision graphs (MDGs). *The Computer Journal*, 47(1):71–84, 2004.

[29] Z. Zhou. *Multiway decision graphs and their applications in automatic formal verification of RTL designs*. PhD thesis, Universite de Montreal, Canada, 1997.

[30] Z. Zhou, X. Song, S. Tahar, E. Cerny, F. Corella, and M. Langevin. Formal verification of the island tunnel controller using multiway decision graphs. In *FMCAD '96: Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, pages 233–247, London, UK, 1996. Springer-Verlag.